

Introspection for C and its Applications to Library Robustness

Manuel Rigger^a, René Mayrhofer^a, Roland Schatz^b, Matthias Grimmer^b, and Hanspeter Mössenböck^a

a Johannes Kepler University Linz, Austria

b Oracle Labs, Austria

Abstract Context: In C, low-level errors, such as buffer overflow and use-after-free, are a major problem, as they cause security vulnerabilities and hard-to-find bugs. C lacks automatic checks, and programmers cannot apply defensive programming techniques because objects (e.g., arrays or structs) lack run-time information about bounds, lifetime, and types.

Inquiry: Current approaches to tackling low-level errors include dynamic tools, such as bounds or type checkers, that check for certain actions during program execution. If they detect an error, they typically abort execution. Although they track run-time information as part of their runtimes, they do not expose this information to programmers.

Approach: We devised an introspection interface that allows C programmers to access run-time information and to query object bounds, object lifetimes, object types, and information about variadic arguments. This enables library writers to check for invalid input or program states and thus, for example, to implement custom error handling that maintains system availability and does not terminate on benign errors. As we assume that introspection is used together with a dynamic tool that implements automatic checks, errors that are not handled in the application logic continue to cause the dynamic tool to abort execution.

Knowledge: Using the introspection interface, we implemented a more robust, source-compatible version of the C standard library that validates parameters to its functions. The library functions react to otherwise undefined behavior; for example, they can detect lurking flaws, handle unterminated strings, check format string arguments, and set *errno* when they detect benign usage errors.

Grounding: Existing dynamic tools maintain run-time information that can be used to implement the introspection interface, and we demonstrate its implementation in Safe Sulong, an interpreter and dynamic bug-finding tool for C that runs on a Java Virtual Machine and can thus easily expose relevant run-time information.

Importance: Using introspection in user code is a novel approach to tackling the long-standing problem of low-level errors in C. As new approaches are lowering the performance overhead of run-time information maintenance, the usage of dynamic runtimes for C could become more common, which could ultimately facilitate a more widespread implementation of such an introspection interface.

ACM CCS 2012

- **Computer systems organization** → **Dependable and fault-tolerant systems and networks**;
- **Software and its engineering** → **Language features**; **Error handling and recovery**; *Software reliability*;

Keywords reflection for C, library robustness, fault tolerance

The Art, Science, and Engineering of Programming

Submitted July 31, 2017

Published December 6, 2017

doi 10.22152/programming-journal.org/2018/2/4



© M. Rigger, R. Mayrhofer, R. Schatz, M. Grimmer, and H. Mössenböck
This work is licensed under a “CC BY 4.0” license.

In *The Art, Science, and Engineering of Programming*, vol. 2, no. 2, 2018, article 4; 31 pages.

1 Introduction

Since the birth of C almost 50 years ago, programmers have written many applications in it. Even the advent of higher-level programming languages has not stopped C's popularity, and it remains widely used as the second-most popular programming language [47]. However, C provides few safety guarantees and suffers from unique security issues that have disappeared in modern programming languages. Buffer overflow errors, where a pointer that exceeds the bounds of an object is dereferenced, are the most serious issue in C [9]. Other security issues include use-after-free errors, invalid free errors, reading of uninitialized memory, and memory leaks. Numerous approaches exist that prevent such errors in C programs by detecting these illegal patterns statically or during run time, or by making it more difficult to exploit them [46, 48, 55]. When an error happens, run-time approaches abort the program, which is more desirable than risking incorrect execution, potentially leaking user data, executing injected code, or corrupting program state.

However, we believe that in many cases programmers could better respond to illegal actions in the application logic if they could use the metadata of run-time approaches (e.g., bounds information) to check invalid actions at run time and prevent them from happening. Library implementers in particular could use it to protect themselves from user input and to compensate for the lack of exception handling in C. For example, if they could check that an access would go out-of-bounds in a server library, they could log the error and ignore the invalid access to maintain availability of the system (as in failure-oblivious computing [35]). If the error happened in the C standard library instead, they could set the global integer variable `errno` to an error code, for example, to `EINVAL` for invalid arguments. Furthermore, a *special value* (such as `-1` or `NULL`) could be returned to indicate that something went wrong. Finally, explicit checks could prevent lurking flaws that would otherwise stay undetected. For example, in the case that a function does not actually access an invalid position in the buffer, bounds checkers cannot detect when an incorrect array size is passed to the function. Using bounds metadata, programmers could validate the passed against the actual array size.

In this paper, we present a novel approach that allows C programmers to query properties of an *object* (primitive value, struct, array, union, or pointer) so that they can perform explicit sanity checks and react accordingly to invalid arguments or states. These properties comprise the bounds of an object, the memory location, the number of arguments of a function with `varargs`, and whether an object can be used in a certain way (e.g., called as a function that expects and returns an `int`). The presented approach is *complementary* to dynamic tools, and does not aim to replace them. Programmers can insert custom input validations and error-handling logic where needed, but the dynamic tool that tracks the exposed metadata still aborts execution for errors that are not handled at the application level. Ultimately, this provides programmers with greater flexibility and increases the robustness of libraries and applications, defined as “[t]he degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions” [21].

As a proof of concept, we implemented the introspection interface for Safe Sulong [32], a bug-finding tool and interpreter with a dynamic compiler for C. Safe Sulong prevents buffer overflows, use-after-free, variadic argument errors, and type errors by checking accesses and aborting execution upon an invalid action. It already maintains relevant run-time information that it can expose to the programmer.

In a case study, we demonstrate how the introspection functions facilitate re-implementing the C standard library (libc) to validate input arguments. We use this libc in Safe Sulong as a source-compatible, more robust drop-in replacement for the GNU C Library. In contrast to the GNU C Library and other implementations, it can detect lurking flaws, handle unterminated strings, check format string arguments, and – instead of terminating execution – set *errno* when errors occur.

A plethora of other dynamic-bug finding tools and runtimes for C exist, and they could expose their run-time data via the introspection functions introduced in this paper. For example, bounds checkers [2, 11, 30, 38] could expose bounds information. Temporal memory safety tools [4, 19, 29, 31, 39, 44] could expose information about memory locations. Variadic argument checkers [3] and type checkers [18, 22] could expose information about variadic arguments and types. There are also combined tools that, for example, provide protection against both out-of-bounds accesses and use-after-free errors [17, 29, 30].

As the overhead of dynamic tools is decreasing [22, 29, 30, 38, 44], they could become standard in production, similar to stack canaries and address space layout randomization [46]. At this point in time, a wider adoption of the introspection functions (or a subset thereof) seems feasible. Additionally, we envisage that dynamic tools available now could distribute specialized libraries that benefit from introspection, as we will demonstrate using Safe Sulong’s libc as an example.

In summary, this paper contributes in the following ways:

- We present introspection functions designed to allow programmers to prevent illegal actions that are specific to C (Section 3).
- We demonstrate how we implemented the introspection functions in Safe Sulong, a bug-finding tool and interpreter with a dynamic compiler for C (Section 4).
- In a case study, we show how using introspection increases the robustness of the C standard library (Section 5).

2 Background

In C, the lack of type and memory safety causes many problems, such as hard-to-find bugs and security issues. Moreover, manual memory management puts the burden of deallocating objects on the programmer. Consequently, C programs are plagued by vulnerabilities that are unique to the language. Faults can invoke undefined behavior, so compiled code can crash, compute unexpected results, and corrupt or read neighboring objects [50, 51]. It is often impossible to design C functions such that they are secure against usage errors, since they cannot validate passed arguments

Introspection for C and its Applications to Library Robustness

or global data. Below we provide a list of errors and vulnerabilities in C programs that we target in this work.

Out-of-bounds errors. Out-of-bounds accesses in C are among the most dangerous software errors [9, 37], since – unlike higher-level languages – C does not specify automatic bounds checks. Further, objects have no run-time information attached to them, so functions that operate on arrays require array-size arguments. Alternatively, they need conventions such as terminating an array by a special value.

Listing 1 shows a typical buffer overflow. The `read_number()` function reads digits entered by the user into the passed buffer `arr` and validates that it does not write beyond its bounds. However, its callee passes `-1` as the `length` parameter, which is (through the `size_t` type) treated as the unsigned number `SIZE_MAX`. Thus, the bounds check is rendered useless, and if the user enters more than nine digits, the `read_number()` function overflows the passed buffer.

A recent similar real-world vulnerability is CVE-2016-3186, where a function in `libtiff` cast a negative value to `size_t`. As another example, in CVE-2016-6823 a function in `ImageMagick` caused an arithmetic overflow that resulted in an incorrect image size. Both faults resulted in buffer overflows.

Memory management errors. Objects that are allocated in different ways (e.g., on the stack or by `malloc()`) have different lifetimes, which influences how they can be used. For example, it is forbidden to access memory after it has been freed (otherwise known as an access to a *dangling pointer*). Other such errors include freeing memory twice, freeing stack memory or static memory, and calling `free()` on a pointer that points somewhere into the middle of an object [29]. Listing 2 shows examples of a use-after-free and a double-free error. Firstly, when `err` is non-zero, the allocated pointer `ptr` is freed and later accessed again as a dangling pointer in `logError()`. Secondly, the code fragment attempts to free the pointer again after logging the error, which results in a double-free vulnerability. C does not provide mechanisms to retrieve the lifetime of an object, which would allow checking and preventing such conditions. Consequently, use-after-free errors frequently occur in real-world code. For example, in CVE-2016-4473 the PHP Zend Engine attempted to free an object that was not allocated by one

■ **Listing 1** Passing `-1` to the `size_t` parameter renders the range check useless and could cause an out-of-bounds error while writing read characters to `arr`

```
1 void read_number(char* arr, size_t length) {
2     int i = 0;
3     if (length == 0) return;
4     int c = getchar();
5     while (isdigit(c) && (i + 1) < length) {
6         arr[i++] = c; c = getchar();
7     }
8     arr[i] = '\0';
9 }
10 // ...
11 char buf[10];
12 read_number(buf, -1);
13 printf("%s\n", buf);
```

■ **Listing 2** Use-after-free error which is based on an example from the CWE wiki

```

1 char* ptr = (char*) malloc(SIZE * sizeof(char));
2 if (err) {
3     abrt = 1; free(ptr);
4 }
5 // ...
6 if (abrt) {
7     logError("operation aborted", ptr); free(ptr);
8 }
9 // ...
10 void logError(const char* message, void* ptr) {
11     logf("error while processing %p", ptr);
12 }

```

of libc’s allocation functions. Other recent examples include a dangling pointer access and a double free error in OpenSSL (CVE-2016-6309 and CVE-2016-0705).

Variadic function errors. Variadic functions in C rely on the programmer to pass a count of variadic arguments or a format string. Furthermore, a programmer must pass the matching number of objects of the expected type. Listing 3 shows an example that uses variadic arguments to print formatted output, similar to C’s `sprintf()` function. It is based on a function taken from the PHP Zend Engine. As arguments, the function expects a format string `fmt`, the variadic arguments `ap`, and a buffer `xbuf` to which the formatted output should be written. To use the function, a C programmer has to invoke a macro to set up and tear down the variadic arguments (respectively `va_start()` and `va_end()`). Using the `va_arg()` macro, `xbuf_format_converter()` can then directly access the variadic arguments. The example shows how a string can be accessed (format specifier `%s`) that is then inserted into the buffer `xbuf`.

The function uses the format string to determine how many variadic arguments should be accessed. For example, for a format string `%s %s` the function attempts to access two variadic arguments that are assumed to have a string type. Accessing a variadic argument via `va_arg()` usually manipulates a pointer to the stack and pops the number of bytes that correspond to the specified data type (`char *` in our example). In a so-called *format string attack*, in which the function reads or writes beyond the stack due to nonexistent arguments, an attacker can exploit the inability of the function to verify the number and the types of the variadic arguments passed [8, 40].

In CVE-2015-8617, this function was the sink of a vulnerability that existed in PHP-7.0.0. The `zend_throw_error()` function called `xbuf_format_converter()` with a message string that was under user control. Consequently, an attacker could use format specifiers without matching arguments to read from and write to memory, and thus execute arbitrary code. As another example, in CVE-2016-4448 a vulnerability in `libxml2` existed because format specifiers from untrusted input were not escaped.

Lack of type safety. Due to the lack of type safety, a programmer cannot verify whether an object referenced by a pointer corresponds to its expected type [22]. Listing 4 demonstrates this for function pointers. The `apply()` function expects a function pointer that accepts and returns an `int`. It uses the function to transform all elements of an array. However, its callee might pass a function that returns a `double`; a call on it would result in undefined behavior. Such “type confusion” cannot be avoided when

Introspection for C and its Applications to Library Robustness

■ Listing 3 Example usage of variadic functions, taken from the PHP Zend Engine

```
1 static void xbuf_format_converter(void *xbuf, const char *fmt, va_list ap) {
2     char *s = NULL;
3     size_t s_len;
4     while (*fmt) {
5         if (*fmt != '%') {
6             INS_CHAR(xbuf, *fmt);
7         } else {
8             fmt++;
9             switch (*fmt) {
10                // ...
11                case 's':
12                    s = va_arg(ap, char *);
13                    s_len = strlen(s);
14                    break;
15                // ...
16            }
17            INS_STRING(xbuf, s, s_len);
18        }
19    }
20 }
```

■ Listing 4 Example of type confusion

```
1 int apply(int* arr, size_t n, int f(int arg1)) {
2     if (f == NULL) return -1;
3     for (size_t i = 0; i < n; i++)
4         arr[i] = f(arr[i]);
5     return 0;
6 }
7
8 double square(int a) { return a * a; }
9
10 apply(arr, 5, square);
```

calling a function pointer, since objects have no types attached that could be used for validation.

Unterminated strings. Unterminated strings are a problem, since the string functions of libc (and sometimes also application code) rely on strings ending with a ‘\0’ (null terminator) character. However, C standard library functions that operate on strings lack a common convention on whether to add a null terminator [28]. Additionally, it is not possible to verify whether a string is properly terminated without potentially causing buffer overreads. Listing 5 shows an example of an unterminated string vulnerability. The read function reads a file’s contents into a string inputbuf. After the call, inputbuf is unterminated if the file was unterminated or if MAXLEN was exceeded. This is likely to cause an out-of-bounds write in strcpy(), since it copies characters to buf until a null terminator occurs. Recent similar real-world vulnerabilities include CVE-2016-7449, where strcpy() was used to copy untrusted (potentially unterminated) input in GraphicsMagick. Further examples are CVE-2016-5093 and CVE-2016-0055, where strings were not properly terminated in the PHP Zend Engine as well as in Internet Explorer and Microsoft Office Excel [27].

Unsafe functions. Some functions in common libraries such as libc have been designed such that they “can never be guaranteed to work safely” [2, 12]. The most prominent

■ **Listing 5** Example fragment that may produce and copy an unterminated string

```
1 read(cfgfile, inputbuf, MAXLEN);
2 char buf[MAXLEN];
3 strcpy(buf, inputbuf);
4 puts(buf);
```

example is the `gets()` function, which reads user input from `stdin` into a buffer passed as an argument. Since `gets()` lacks a parameter for the size of the supplied buffer, it cannot perform any bounds checking and overflows the buffer if the user input is too large. Although C11 replaced `gets()` with the more robust `gets_s()` function, legacy code might still require the unsafe `gets()` function. In general, functions that lack size arguments – which prevents safe access to arrays – cannot be made safe without breaking source and binary compatibility.

3 Introspection Functions

To enable C programmers to validate arguments and global data, we devised introspection functions to query properties of C objects and the current function (see Appendix B). These functions allow programmers only to inspect objects and not to manipulate them; therefore, the presented functions are not a full reflection interface.

We designed these functions specifically to provide users with the ability to prevent buffer overflow, use-after-free, and other common errors specific to C. Through introspection, programmers can validate certain properties (memory location, bounds, and types) before performing an operation on an object. Additionally, introspection allows the number of variadic arguments passed to be queried and their types to be validated.

We built introspection based on several *introspection primitives*. These primitives are a minimal set of C functions that require run-time support. We also designed *introspection composites*, which are implemented as normal C functions and are based on the introspection primitives or on other composites. The introspection functions that we expose to the programmer contain both selected primitives and composites. We hereafter denote internal functions that are private to an implementation with an underscore prefix.

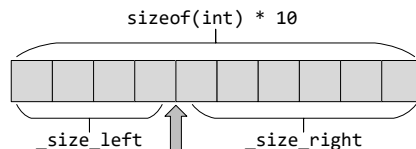
3.1 Object Bounds

Most importantly, we provide functions that enable the programmer to perform bounds checks before accessing an object. Simply providing a function that returns the size of an object is insufficient, since a pointer can point to the middle of an object. Instead, we require the runtime to provide two functions to return the space (in bytes) to the left and to the right of a pointer target: `_size_left()` and `_size_right()`. Their result is only defined for *legal* pointers, which we define as pointers that point to valid objects (not `INVALID`, see Section 3.2).

Introspection for C and its Applications to Library Robustness

■ **Listing 6** Example of how to query the space to the left and to the right of a pointer

```
1 int *arr = malloc(sizeof(int) * 10);
2 int *ptr = &arr[4];
3 printf("%d\n", size_left(ptr)); // prints 16
4 printf("%d\n", size_right(ptr)); // prints 24
```



■ **Figure 1** Memory Layout of the Example in Listing 6

■ **Listing 7** Implementation of `size_left()` using the functions `location()`, `_size_left()`, and `_size_right()`

```
1 long size_left(const void *ptr) {
2     if (location(ptr) == INVALID) return -1;
3     bool inBounds = _size_right(ptr) >= 0 && _size_left(ptr) >= 0;
4     if (!inBounds) return -1;
5     return _size_left(ptr);
6 }
```

Listing 6 illustrates the function return values when passing a pointer to the middle of an integer array to these functions. For the pointer to the fourth element of the ten-element integer array, `_size_left()` returns `sizeof(int) * 4`, and `_size_right()` returns `sizeof(int) * 6`. Figure 1 shows the corresponding memory layout. On an architecture where an `int` is four bytes in size the functions return 16 and 24, respectively.

We do not expose these two functions to the programmer, but base the composite functions `size_left()` and `size_right()` on them, which return -1 if the passed argument is not a legal pointer or out of bounds. Listing 7 shows the implementation of `size_left()`. Using `location()`, the function first checks that the pointer is legal (see Section 3.2). It then checks that the spaces to the left and to the right of the pointer are not negative, that is, the pointer is in bounds. If both checks are passed, the function returns the space to the left of the pointer using `_size_left()`; otherwise, it returns -1.

Listing 8 shows how using `size_right()` improves `read_number()`'s robustness (see Listing 1): If `arr` is a valid pointer but points to memory that cannot hold length chars, we can prevent the out-of-bounds access by aborting the program. Note that the check also detects lurking bugs, since it aborts even if fewer than length characters are read. If `arr` is not a valid pointer, the return value of `size_right()` is -1.

3.2 Memory Location

Querying the memory location of an object (e.g., stack, heap, global data) allows a programmer to obtain information about the lifetime of an object. For example, it enables programmers to prevent use-after-free errors by detecting whether an object has already been freed. Another use case is validating that no stack memory is

- **Listing 8** By using the `size_right()` function we can avoid out-of-bounds accesses in `read_number()`

```

1 void read_number(char* arr, size_t length) {
2     int i = 0;
3     if (length == 0) return;
4     if (size_right(arr) < length) abort();
5     // ...
6 }

```

- **Listing 9** Example of how the `location()` enum constants relate to objects in a program

```

1 int a; // location(&a) returns STATIC for global objects
2 void func() {
3     static int b; // location(&b) returns STATIC for static local objects
4     int c; // location(&c) returns AUTOMATIC for stack objects
5     int* d = malloc(sizeof(int) * 10);
6     // location(&d) returns DYNAMIC for heap objects
7     free(d); // location(&d) returns INVALID for freed objects
8 }

```

- **Listing 10** By using `location()` and `_size_left()` we can check whether an object can be freed

```

1 bool freeable(const void *ptr) {
2     return location(ptr) == DYNAMIC && _size_left(ptr) == 0;
3 }

```

returned by a function. A programmer can also check whether a location refers to dynamically allocated memory to ensure that `free()` can be safely called on it. For this purpose, we provide the function `location()`, which determines where an object lies in memory.

The function returns one of the following enum constants:

- `INVALID` locations denote `NULL` pointers or deallocated memory (freed heap memory or dead stack variables). Programs must not access such objects.
- `AUTOMATIC` locations denote non-static stack allocations. Functions must not return allocated stack variables that were declared in their scope, since they become `INVALID` when the function returns. Further, stack variables must not be freed.
- `DYNAMIC` locations denote dynamically allocated heap memory created by `malloc()`, `realloc()`, or `calloc()`. Only memory allocated by these functions can be freed.
- `STATIC` locations denote statically allocated memory such as global variables, string constants, and static local variables. Static compilers usually place such memory in the text or data section of an executable. Programs must not free statically allocated memory.

Listing 9 shows how differently allocated memory relates to the enum constants used by `location()`.

We provide the function `freeable()`, which is based on `location()`, to conveniently check whether an allocation can be freed. As Listing 10 demonstrates, a `freeable` object's location must be `DYNAMIC`, and its pointer must point to the beginning of an object. Listing 11 shows how we can use the `freeable()` function to improve the

Introspection for C and its Applications to Library Robustness

■ **Listing 11** By using the `freeable()` function we can avoid double-free errors

```
1 char* ptr = (char*) malloc(SIZE * sizeof(char));
2 if (err) {
3     abrt = 1;
4     if (freeable(ptr)) free(ptr);
5 }
6 // ...
7 if (abrt) {
8     logError("operation aborted", ptr);
9     if (freeable(ptr)) free(ptr);
10 }
```

■ **Listing 12** By using the `location()` function we can avoid use-after-free errors

```
1 void logError(const char* message, void* ptr) {
2     if (location(ptr) == INVALID)
3         log("dangling pointer passed to logError!");
4     else
5         logf("error while processing %p", ptr);
6 }
```

robustness of the code fragment shown in Listing 2. It ensures that freeing the pointee is valid, and thus prevents invalid free errors, such as double freeing of memory. Nonetheless, the `logError()` function may receive a dangling pointer as an argument. To resolve this, we can check in `logError()` whether the pointer is valid (see Listing 12).

Note that some libraries, such as OpenSSL, use custom allocators to manage their memory. Custom allocators are outside the scope of this paper, but could be supported by providing source-code annotations for allocation and free functions; this information could then be used by the runtime to track the memory. The annotations for the allocation functions would need to specify how to compute the size of the allocated object, and the location of the allocated memory. Additionally, it might be desirable to add further enum constants, for example, for shared, file-backed, or protected memory. We omitted additional constants for simplicity.

3.3 Type

We provide a function that allows the programmer to validate whether an object is *compatible with* (can be treated as being of) a certain type. Such a function enables programmers to check whether a function pointer actually points to a function object (and not to a long, for example) and whether it has the expected function signature. As another example, programmers can use the function as an alternative to `size_right()` and `size_left()` to verify that a pointer of a certain type can be dereferenced.

C has only a weak notion of types, which makes it difficult to design expressive type introspection functions. For example, it is ambiguous whether a pointer of type `int*` that points to the middle of an integer array should be considered as a pointer to an integer or as a pointer to an integer array. Another example is heap memory, which lacks a dynamic type; although programmers usually coerce them to the desired type, objects of different types can be stored. Even worse, when writing to memory, objects

■ **Listing 13** By using `try_cast()` we can ensure that we can perform an indirect call on the function pointer in `apply()`

```

1 int apply(int* arr, size_t n, int f(int arg1)) {
2     if (size_right(arr) < sizeof(int) * n || try_cast(&f, type(f)) == NULL)
3         return -1;
4     for (size_t i = 0; i < n; i++)
5         arr[i] = f(arr[i]);
6     return 0;
7 }

```

can be partially overwritten; for instance, half of a function pointer can be overwritten with an integer value, which makes it difficult to decide whether the pointer is still a valid function pointer.

Instead of assuming that a memory region has a specific type, we designed a function that allows the programmer to check whether the memory region is compatible with a certain type (similar to [22]). The `try_cast()` function expects a pointer to an object as the first argument and tries to cast it to the Type specified by the second argument. If the runtime determines that the cast is possible, it returns the passed pointer, and `NULL` otherwise. The cast is only possible if the object can be read, written to, or called as the specified type.

The Type object is a recursive struct which makes it possible to describe nested types (known as type expressions [1]). For example, a function pointer with an `int` parameter and `double` as the return type can be represented by a tree of three Type structs. The root struct specifies a function type and references a struct with an `int` type as the argument type as well as a struct with a `double` type as the return type. Since manually constructing Type structs is tedious, we specified the *optional* operator `type()`. As an argument, it requires the expression *example value*, whose declared type is returned as a Type run-time data structure. Since the declared type is a compile-time property, we want to resolve the `type()` operator during compile time; consequently, the programmer cannot take `type()`'s address and call it indirectly. The operator is similar to the GNU C extension `typeof`, which yields a type that can be used directly in variable declarations or casts.

Listing 13 shows how the type introspection functions make the function `apply()` (see Listing 4) more robust: `apply()` uses `try_cast()` to check whether the runtime can treat its first argument as the specified function pointer. Its second argument is the Type object that the type operator constructs from the declared function pointer type. The `try_cast()` function returns the first argument if it is compatible with the specified function pointer type; otherwise, it returns `NULL`. In addition to preventing the calling of invalid function pointers, `apply()` prevents out-of-bounds accesses by validating the array size.

The `try_cast()` function is similar to C++'s `dynamic_cast()`. However, we want to point out that C++'s `dynamic_cast()` works only for class checks (which are well-defined), while our approach works for all C objects. We believe that the exact semantics of `try_cast()` should be implementation-defined, since run-time information could differ between implementations. For example, depending on the runtime's knowledge of data execution prevention, it might either allow or reject the cast of a non-executable

Introspection for C and its Applications to Library Robustness

■ **Listing 14** By using `count_varargs()` and `get_varargs()` we can use variadics in a robust way

```
1 double avg(int count, ...) {
2     if (count == 0 || count != count_varargs())
3         return 0;
4     int sum = 0;
5     for (int i = 0; i < count; i++) {
6         int *arg = get_vararg(i, type(&sum));
7         if (arg == NULL) return 0;
8         else          sum += *arg;
9     }
10    return (double) sum / count;
11 }
```

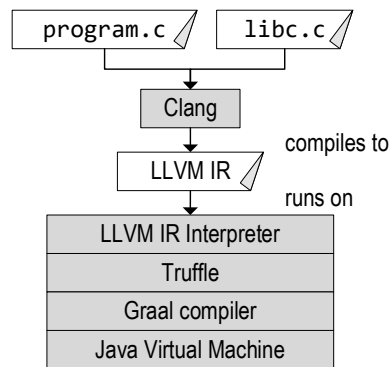
char array filled with machine instructions to a function pointer. Further, different use cases exist, and a security-focused runtime might have more sources of run-time information and be more restrictive than a performance-focused runtime. For example, a traditional runtime would (for compatibility) allow dereferencing a hand-crafted pointer as long as it corresponds to the address of an object, while a security-focused runtime could disallow it. Thus, depending on the underlying runtime, compiler, and ABI, `try_cast()` can return different results.

3.4 Variadic Arguments

Our introspection interface provides macros to query the number of variadic arguments and enables programmers to access them in a type-safe way. They are implemented as macros and not as functions, since they need to access the current function's variadic arguments. The introspection macros make using variadic functions more robust and are, for example, effective in preventing format string attacks [8].

Querying the number of variadic arguments can be achieved by calling `count_varargs()`. The standard `va_arg()` macro reads values from the stack while assuming that they correspond to the user-specified type. As a robust alternative, introspection composites can use `_get_vararg()` to access the passed variadic arguments directly by an argument index. To access the variadic arguments in a type-safe way, we introduced the `get_vararg()` macro, which is exposed to the programmer and expects a type that it uses to call `try_cast()`. Listing 14 shows an example of a function that computes the average of `int` arguments. It uses `count_varargs()` to verify the number of variadic arguments and ensures that the i^{th} argument is in fact an `int` by calling `get_vararg()` with `type(&sum)`. If an unexpected number of parameters or an object with an unexpected type is passed, the function returns `0`.

For backwards compatibility, we used the introspection intrinsics to make the standard `vararg` macros (`va_start()`, `va_arg()`, and `va_end()`) more robust. Firstly, `va_start()` initializes the retrieval of variadic arguments. We modified it such that it allocates a struct (using the `alloca()` stack allocation function) and populates it using `_get_vararg()` and `count_varargs()`. The struct comprises the number of variadic arguments, an array of addresses to the variadic arguments, and a counter to index them. Secondly, `va_arg()` retrieves the next variadic argument. We modified it such that it checks that the counter does not exceed the number of arguments, increments the counter, in-



■ **Figure 2** Overview of Safe Sulong

dexes the array, and casts the variadic argument to the specified type using `try_cast()`. If the cast succeeds, the argument is returned; otherwise a call to `abort()` exits the program. Finally, `va_end()` performs a cleanup of the data initialized by `va_start()`. We modified it such that it resets the variadic arguments counter.

Using the enhanced `vararg` macros improves the robustness of the `xbuf_format_converter()` function (see Listing 3), since the number of format specifiers must match the number of arguments, thus making it impossible to exploit the function through format string attacks. Note that the modified standard macros abort when they process invalid types or an invalid number of arguments, whereas the intrinsic functions allow programmers to react to invalid arguments in other ways.

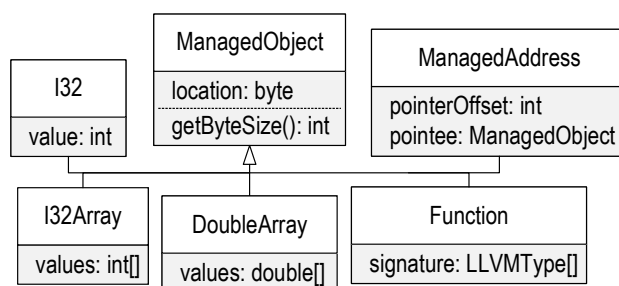
4 Implementation

We implemented the introspection primitives in Safe Sulong [32], which is an execution system and bug-finding tool for low-level languages such as C. At its core is an interpreter written in Java that runs on top of the JVM. Although this setup is not typical for running C, it is a good experimentation platform because the JVM (and thus also Safe Sulong) already maintains all the run-time metadata that we want to expose. If exposing introspection primitives turns out to be useful for Safe Sulong, similar mechanisms could also be implemented for other runtimes (e.g., those of static compilation approaches). Unlike its counterpart Native Sulong [33], Safe Sulong uses Java objects to represent C objects. By relying on Java’s bounds and type checks, Safe Sulong efficiently and reliably detects out-of-bounds accesses, use-after-free, and invalid free. When detecting such an invalid action, it aborts execution of the program. Section 4.1 gives an overview of the system, and Section 4.3 describes how we implemented the introspection primitives.

4.1 System Overview

Figure 2 shows the architecture of Safe Sulong, which comprises the following components:

Introspection for C and its Applications to Library Robustness



■ **Figure 3** Diagram of the ManagedObject Hierarchy

Clang. Safe Sulong executes LLVM Intermediate Representation (IR), which represents C functions in a simpler, but lower-level format. LLVM is a flexible compilation infrastructure [25], and we use LLVM's front end Clang to compile the source code (libraries and the user application) to the IR.

LLVM IR. LLVM IR retains all C characteristics that are important for the content of this paper. It can, for instance, contain external function definitions and function calls. By executing LLVM IR, Safe Sulong can execute all languages that can be compiled to this IR, including C++ and Fortran. Using binary translators that convert binary code to LLVM IR even allows programs to be executed without access to their source code. For example, MC-Semantics [10] and QEMU [6] support x86, and LLBT [41] supports the translation of ARM code. Binary libraries that are converted to LLVM IR can then profit from the enhanced libraries that Safe Sulong can execute, such as our enhanced libc.

Truffle. The LLVM IR interpreter is based on Truffle [53]. Truffle is a language implementation framework written in Java. To implement a language, a programmer writes an Abstract Syntax Tree (AST) interpreter in which each operation is implemented as an executable node. Nodes can have children that parent nodes can execute to compute their results.

Graal. Truffle uses Graal [54], a dynamic compiler, to compile frequently executed Truffle ASTs to machine code. Graal applies aggressive optimistic optimizations based on assumptions that are later checked in the machine code. If an assumption no longer holds, the compiled code *deoptimizes* [20], that is, control is transferred back to the interpreter and the machine code of the AST is discarded.

LLVM IR Interpreter. The LLVM IR interpreter forms the core of Safe Sulong; it executes both the user application and the enhanced libc. First, a front end parses the LLVM IR and constructs a Truffle AST for each LLVM IR function. Then, the interpreter starts executing the main function AST, which can invoke other ASTs. During execution, Graal compiles frequently executed functions to machine code.

JVM. The system can run efficiently on any JVM that implements the Java-based JVM compiler interface (JVMCI [36]). JVMCI supports Graal and other compilers written in Java.

4.2 Introspection Primitives and Other Functions

While the majority of Safe Sulong’s libc is implemented in C, the introspection primitives (and a core API, similar to system calls) are implemented directly in Java. Both are ultimately represented using executable ASTs, which are stored in a symbol table created prior to program execution. For functions contained in the LLVM IR file, the parser constructs the AST nodes from the instructions denoted in the LLVM IR function. For introspection primitives, we implemented special nodes that have no equivalent bitcode instruction (see Section 4.3). During execution, Safe Sulong looks up the AST in the symbol table using the function name. From the runtime’s perspective, the implementation of that function is transparent.

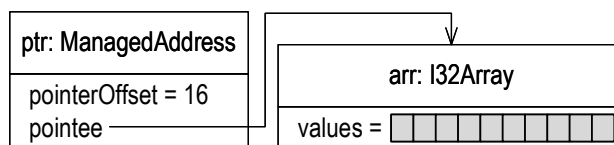
4.3 Objects and Introspection

The LLVM IR interpreter uses Java objects instead of native memory to represent LLVM IR objects (and thus C objects). Figure 3 illustrates its type hierarchy. Every LLVM IR object is a `ManagedObject` which has subclasses for the different types. For example, an `int` is represented by an `I32` object, which stores the `int`’s value in the value field. Similarly, there are subclasses for arrays, functions, pointers, structs, and other types. Note that we have previously described a similar object hierarchy for the implementation of a *Lenient C* dialect and how certain corner cases are supported (e.g., deriving pointers from integers) [34]. In the introspection implementation, we needed to expose properties of these Java objects to the programmer:

Bounds. The `ManagedObject` class provides the method `getByteSize()`, which returns the size of an object. Safe Sulong represents pointers as objects of a `ManagedAddress` class that holds a reference to the pointee and a pointer offset that is updated through pointer arithmetics (pointee and `pointerOffset`). For example, for the pointer to the 4th element of an integer array in Listing 6, the `pointerOffset` is 16, and pointee references an `I32Array` that holds a Java `int` array (see Figure 4). If a program were to dereference the pointer, the interpreter would compute `pointerOffset / sizeof(int)` to index the array. We implemented the `size_right()` function by `ptr.pointee.getByteSize() - ptr.pointerOffset`.
Memory location. Although `ManagedObjects` live on the Java heap, the `location()` function needs to return their *logical* memory location. This location is stored in a field of the `ManagedObject` class. Depending on whether an object is allocated through `malloc()`, as a global variable, as a static local variable, or as a constant, we assign a different flag to its location field; calls to `free()` and deallocation of automatic variables assign `INVALID`. For instance, for an integer array that lives on the stack, the interpreter allocates an `I32Array` and assigns `AUTOMATIC` to its location. After leaving the function scope, its location is updated to `INVALID`. When the `location()` function is called with a pointer to the integer array, it returns the location field’s value.

Type. For implementing the `try_cast()` function, we check if the type of the passed object (given by its Java class) is compatible with the type specified by the `Type` struct. For example, to check whether we can call a pointer as a function with a certain signature, we first compare the passed pointer with a `Type` that describes this signature.

Introspection for C and its Applications to Library Robustness



■ **Figure 4** Representation of a pointer to the 4th element of an int array

If the pointer references a Safe Sulong object of type Function, the argument and return types are compared. This is possible because Function objects retain run-time information about their arguments and return types, which can be retrieved via the method `getSignature()`.

Variadic arguments. In Safe Sulong, a caller explicitly passes its arguments as an object array (i.e., a Java array of known length) to its callee. Based on the function signature and the object array, the callee can count the variadic arguments to implement `count_varargs()` and extract them to implement `_get_vararg()`.

5 Case Study: Safe Sulong's Standard Library

We implemented an enhanced libc for Safe Sulong. This libc uses introspection for checks that make it more robust against usage errors and attacks. For instance, its functions identify invalid parameters that would otherwise cause out-of-bounds accesses or use-after-frees. In such a case, the functions return special values to indicate that something went wrong, and then set `errno` to an error code. However, for functions for which no special value can be returned (e.g., because the return type is void), setting `errno` would be meaningless, since functions are allowed to change `errno` arbitrarily even if no error occurred. In these cases, the functions still attempt to compute a meaningful result. Such behavior is compliant with the C standards, since we prevent illegal actions with undefined behavior that could crash the program or corrupt memory.

For applications and libraries that run on Safe Sulong, the distribution format is LLVM IR and not executable code. Our standard library improvements are binary-compatible at the IR level, which means that users do not have to recompile their applications when using our enhanced libc. In addition, this standard library is source-compatible, so a user is not required to change the program when using it. Below, we give an overview of our enhanced library functions:

String functions. We made all functions that operate on strings (`strlen()`, `atoi()`, `strcmp()`, `printf()`, etc.) more robust by computing meaningful results even when a string lacks a null terminator. They do not read or write outside the boundaries of unterminated strings, which makes them robust against common string vulnerabilities. The functions increase availability of the system, since unterminated strings passed to libc do not cause crashes. Note that when a function outside libc relies on a terminated string, it will still trigger an out-of-bounds access and cause Safe Sulong to abort execution.

■ **Listing 15** Robust implementation of `strlen()` that also works for unterminated strings

```

1 size_t strlen(const char *str) {
2     size_t len = 0;
3     while (size_right(str) > 0 && *str != '\0') {
4         len++; str++;
5     }
6     return len;
7 }

```

Thus, increased availability does not harm confidentiality (e.g., by leaking data of other objects) and integrity (e.g., by overwriting other objects).

For instance, Listing 15 shows how we improved `strlen()` by preventing buffer overflows when iterating over a string, and by improving the handling of non-legal pointers (where `size_right()` returns `-1`). For terminated strings, `strlen()` iterates until the first `'\0'` character to return the length of the string. For unterminated strings, the function cannot return `-1` to indicate an error, since `size_t` is unsigned, so we also do not set `errno`. Instead, it iterates until the end of the buffer and returns the size of the string until the end of the buffer.

The enhanced string functions also allow execution of the code fragment in Listing 5. Even though the source string may be unterminated, `strcpy()` will not produce an out-of-bounds read, since it stops copying when reaching the end of the source or destination buffer. The call to `puts()` also works as expected, and prints the unterminated string.

Functions that free memory. We made functions that free memory (`realloc()` and `free()`) more robust by checking whether their argument can safely be freed using `freeable()`. In Safe Sulong, `malloc()` is written in Java and allocates a Java object. By using the introspection functions we were able to conveniently and robustly implement `realloc()` in C without having to maintain a list of allocated and freed objects.

Format string functions. We made input and output functions that expect format strings more robust. Examples are the `printf()` functions (`printf()`, `fprintf()`, `sprintf()`, `vfprintf()`, `vprintf()`, `vsprintf()`, `vsnprintf()`, `vsprintf()`) and the `scanf()` functions (`scanf()`, `fscanf()`, etc.). These functions expect format strings that contain format specifiers, and matching arguments that are used to produce the formatted output. Since the functions are variadic, we used `count_varargs()` to add checks that verify that the number of format specifiers is equal to the actual number of arguments. Further, the functions use `get_vararg()` to verify the argument types. This prevents format-string vulnerabilities and out-of-bounds reads in the format string, as demonstrated in the implementation of `strlen()`.

Higher-order functions. We enhanced functions that receive function pointers such as `qsort()` and `bsearch()`. Listing 16 shows how `qsort()` can use `try_cast()` to verify that `f` is a function pointer that is compatible with the specified signature. Furthermore, the functions verify that no memory errors, such as buffer overflows, can occur.

`gets()` and `gets_s()`. While C11 replaced the `gets()` function with `gets_s()`, Safe Sulong can still provide a robust implementation for `gets()` (see Listing 17). Since `size_right()` can determine the size of the buffer to the right of the pointer, we can call it and use the returned size as an argument to the more robust `gets_s()` function. If the pointer

Introspection for C and its Applications to Library Robustness

- **Listing 16** Robust `qsort()` implementation that checks whether it can call the supplied function pointer

```
1 void qsort(void *base, size_t nitems, size_t size, int (*f)(const void *, const void*)) ←  
    ↪ {  
2     int (*verifiedPointer)(const void *, const void*) = try_cast(&f, type(f));  
3     if (size_right(base) < nitems * size || verifiedPointer == NULL) errno = EINVAL;  
4     else {  
5         // qsort implementation  
6     }  
7 }
```

- **Listing 17** Robust implementation of `gets()` that uses the more robust `gets_s()` in its implementation

```
1 char *gets(char *str) {  
2     int size = size_right(str);  
3     return gets_s(str, size == -1 ? 0 : size);  
4 }
```

- **Listing 18** Robust implementation of `gets_s()` that verifies the passed size argument

```
1 char *gets_s(char *str, rsize_t n) {  
2     if (size_right(str) < (long) n) {  
3         errno = EINVAL; return NULL;  
4     } else {  
5         // original code  
6     }  
7 }
```

is not legal, we pass 0, which `gets_s()` handles as an error. We also made `gets_s()` more robust against erroneous parameters (see Listing 18). By using `size_right()` we can validate that the size parameter `n` is at least as large as the remaining space right of the pointer. The check prevents buffer overflows for `gets()` and `gets_s()`, and also passing of dead stack memory or freed heap memory.

6 Related Work

C Memory safety approaches. For decades, academia and industry have been coming up with approaches to tackling memory errors in C. Thus, there is a vast number of approaches that deal with these issues, both static and run-time approaches, both hardware- and software-based. We consider our approach as a run-time approach, since the checks (specified by programmers in their programs) are executed during run time. The literature provides a historical overview of memory errors and defense mechanisms [48], an investigation of the weaknesses of current memory defense mechanisms including a general model for memory attacks [46], and a survey of vulnerabilities and run-time countermeasures [55]. Using introspection to prevent memory errors is a novel approach that is complementary to existing approaches because the programmer can check for and prevent an invalid action; if the check is

omitted and an invalid access occurs, an existing memory safety solution could still prevent the access.

Run-time types for C. `libcrunch` [22] is a system that detects type-cast errors at run time. It is based on `liballocs` [23], a run-time system that augments Unix processes with allocation-based types. `libcrunch` provides an `__is_a()` introspection function that exposes the type of an object. It uses this function to validate type casts and issues a warning on unsound casts. In contrast to our approach, `libcrunch` checks for invalid casts automatically, so the `__is_a()` function is not exposed to the programmer, nor are there other introspection functions. However, we believe that the system could be extended to provide additional run-time information that could be used to implement the introspection primitives. Typical overheads of collecting and using the type information are between 5-35%, which demonstrates that introspection functions are feasible in static compilation approaches.

Failure-oblivious computing. Failure-oblivious computing [35] is a technique that enables servers to continue their normal execution path in the presence of memory errors. Instead of aborting the program, invalid writes are discarded, and for invalid reads values are manufactured. Note that this approach is automatic, since the compiler inserts checks and continuation code where memory errors can occur. Failure-oblivious computing would, for example, work well for `strlen` by manufacturing the value zero when the NULL terminator is missing and the read runs over the buffer end. However, returning zero for out-of-bounds accesses does not work in general; for example, when the loop's exit condition checks if the array element is `-1`, failure-oblivious computing approaches could run into an endless loop. In contrast, using our introspection technique, programmers can take into account the semantics of a function to prevent such situations. Additionally, introspection can also be used for bug-finding (not only to increase availability), for example, by checking if the actual buffer length corresponds to the expected buffer length in functions like `gets_s`.

Static vulnerability scanners. Static vulnerability scanners identify calls to unsafe functions such as `gets()` depending on a policy specified in a vulnerability database [49]. Such approaches must decide conservatively whether a call is allowed, unlike our approach, which validates parameters at run-time through introspection. Nowadays, most compilers issue a warning when they identify a call to an unsafe function such as `gets()`, but not necessarily for other, slightly safer functions, such as `strcpy()`.

Fault injection to increase library robustness. Fault injection approaches generate a series of test cases that exercise library functions in an attempt to trigger a crash in them. `HEALERS` [13, 14] is an approach that, after identifying a non-robust function, automatically generates a wrapper that sits between the application and its shared libraries to handle or prevent illegal parameters. To check the bounds of heap objects passed to the functions, the approach instruments `malloc()` and stores bounds information. In contrast to our solution, the approaches above support pre-compiled libraries. However, they can generate wrapper checks only where run-time information is explicitly available in the program. Additionally, they prevent the programmer from specifying the action in case of an error, and always set `errno` and return an error code.

Introspection for C and its Applications to Library Robustness

Detecting API misuses. APISan [56] is a tool for finding API usage errors, such as cryptographic protocol API misuses, but also integer overflows, NULL dereferences, memory leaks, incorrect return values, format string vulnerabilities, and wrong arguments. It is based on the idea that the dominant usage pattern of an API across several projects indicates its correct use. APISan is implemented by gathering execution traces using symbolic execution, from which it infers correct API usages; deviating patterns are potential API misuses. While this approach aims to identify incorrect use of libraries, our approach aims to make library functions more robust.

Replacing (parts of) libc. SFIO [24] is a libc replacement and addresses several of its problems. It mainly improved completeness and efficiency, but it also introduced safer routines for functions that operate on format strings. Additionally, the SFIO standard library functions are more consistent in their arguments and argument order, and thus less error-prone than some of the libc functions. In [28], the less error-prone `strncpy()` and `strncat()` functions were presented as replacements for the `strcpy()` and `strcat()` functions. Unlike our improved C standard library, these approaches lack source compatibility.

Safer implementation of library functions. To prevent format string vulnerabilities in the `printf` family of functions, FormatGuard [8] uses the preprocessor to count the arguments to variadic functions during compile time and checks that the number complies with the actual number at run time. FormatGuard replaces the `printf` functions in the C standard library with more secure versions while retaining compatibility with most programs. From a user perspective, FormatGuard is similar to Safe Sulong's standard library, in that both provide more robust C standard library functions. While our approach works only for runtimes that implement the introspection primitives, StackGuard works for arbitrary compilers and runtimes. However, our approach can also verify bounds, memory location, and types of objects.

Restricting buffer overflows in library functions. Libsafe [2] replaces calls to unsafe library functions (such as `strcpy()` and `gets()`) with wrappers that ensure that potential buffer overflows are contained within the current stack frame. It can prevent only stack buffer overflows, since it checks that write accesses do not extend beyond the end of the buffer's stack frame. In contrast, approaches exist that protect only against heap buffer overflows caused by C standard library functions [15]. By intercepting C standard library calls, the approach keeps track of heap memory allocations and performs bounds checking before calling the C standard library functions that operate on buffers. Both approaches work with any existing pre-compiled library, but do not protect against all kinds of buffer overflows. With our approach, a programmer can implement checks that prevent both heap and stack overflows, and use the introspection interface to also prevent use-after-free and other errors.

Reflection for C. Higher-level languages such as Java or C# throw exceptions when encountering out-of-bounds accesses and other errors. Exception handling is a more expressive approach than explicitly checking for invalid accesses in advance, since it separates the two concerns in the program. Some approaches introduced mechanisms to raise and catch exceptions in C [16, 26]. However, these approaches do not describe

how invalid memory errors could be caught and exposed to the programmer as an exception.

7 Discussion

Advantages over existing tools. We assume that introspection is exposed by a runtime that automatically aborts when detecting an error (e.g., an out-of-bounds access). In this scenario, using introspection allows programmers to override the default behavior of aborting the program by checking for invalid states and by reacting to them before the failure occurs. Even if checks are omitted, the runtime aborts execution in case of an error. Additionally, introspection can be used to check for faults that might not result in errors during run time. While adding these checks does not come for free (i.e., they require programming effort), we believe that they can be useful at boundaries of shared libraries, and at the boundaries of subcomponents within a project.

Adoption of introspection. Two of the C/C++ tenets are that “you don’t pay for what you don’t use” [45] and to “trust the programmer” [5]. Hence, programmers often eschew checks even if they are possible without introspection functions [14]. An open question is thus whether C programmers would use introspection if they had access to it. We believe that there is a need for the safe execution of legacy C code (at the expense of performance) as an alternative to porting programs to safer languages. It has yet to be determined which of the introspection functions are useful in practice (e.g., by conducting a case study on real-world programs). We believe that functions such as `size_right()` are easy to understand and use, and could prevent common errors in practice. In contrast, grasping the semantics of `try_cast()` is more difficult because C does not have a strong notion of typing, and use cases for it are also rare; consequently, it would probably be used less often.

Safer languages. Since using introspection requires changes to the source code, a question is whether a library should not simply be rewritten in some other systems programming language, such as Rust or Go, that approach the performance of C while being safe. First, preventing out-of-bounds accesses or use-after-free errors can already be prevented by using special runtimes without rewriting the project in a safer language (e.g., using `AddressSanitizer` [38] or `SoftBound+CETS` [29, 30]). However, our approach goes beyond these guarantees by allowing the programmer to handle errors in customized ways. Second, the effort required to rewrite an application would simply be too high for many real-world applications. In contrast, incrementally adding checks to an existing code base is less work.

Legacy code. Our approach also brings benefits for legacy applications, namely when a commonly used shared library is modified to employ introspection for additional checks: For example, there are legacy applications that use the insecure `gets()` libc function. Using our approach, a safe implementation of `gets()` can be provided if the runtime implements the introspection interface and libc uses it to query the length of the buffer. Thus, availability or security of legacy code can be improved simply by employing a libc that inserts additional checks enabled by introspection. In contrast,

Introspection for C and its Applications to Library Robustness

when reimplementing `libc` in a safer language, the function `gets()` cannot be made safe, as a buffer allocated by C code has no bounds information attached to it.

Static compilation. Introspection requires information about run-time properties of objects in the program. While interpreters and virtual machines often maintain this information, runtimes that execute native programs compiled by static compilers such as Clang or GCC do not. We want to point out that debug metadata (obtained by compiling with the `-g` flag) cannot provide per-object type information needed for introspection. However, it has been shown that per-object information (such as types) can be added at low cost to static compilation approaches [22] and hence make implementing the introspection functions in their runtimes feasible. As part of future work, we intend to implement introspection primitives using tools based on a static compilation model.

Partial metadata availability. While designing the interface, we assumed that a tool that implements introspection maintains all relevant metadata. However, some runtimes maintain only a subset of it; for example, bounds checkers track bounds information and can implement only `_size_left()` and `_size_right()`. Custom memory allocators that track heap allocations can implement only a subset of the function location(). It has yet to be investigated how code can benefit from runtimes that implement only parts of the interface. A compile-time approach would involve checking introspection features using preprocessor directives. Another approach would involve structuring the checks such that they do not fail when an introspection function returns a default value that indicates that the corresponding feature is unsupported.

Performance measurement. The focus of this work was on evaluating the usefulness of exposing introspection functions to library writers. We did not invest much time in optimizing the peak performance of our approach in Safe Sulong. Thus, we show its performance only on a small set of microbenchmarks for which we used our enhanced `libc` (see Appendix A). As part of future work, we want to extend Safe Sulong's completeness to execute larger benchmarks, such as SPEC INT [7].

8 Conclusion

We have presented an introspection interface for C that programmers can use to make libraries more robust. The introspection functions expose properties of objects (bounds, memory location, and type) as well as properties of variadic functions (number of variadic arguments and their types). We have described an implementation of the introspection primitives in Safe Sulong, a system that provides memory-safe execution of C code. However, our approach is not restricted to Safe Sulong; many dynamic bug-finding tools and runtimes exist that could implement (a subset of) the introspection interface. The approach is complementary to existing memory safety approaches, as programmers can use it to react to and prevent errors in the application logic. Finally, we have shown how we used the introspection interface to implement an enhanced, source-compatible C standard library.

Acknowledgements We thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. We also thank Ingrid Abfalter, whose proofreading and editorial assistance greatly improved the manuscript. We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at Johannes Kepler University Linz for their support and contributions.

References

- [1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.
- [2] Arash Baratloo, Navjot Singh, and Timothy Tsai. “Libsafe: Protecting critical elements of stacks”. In: *White Paper* (1999). <http://www.research.avayalabs.com/project/libsafe>.
- [3] Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. “Venerable Variadic Vulnerabilities Vanquished”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/biswas>.
- [4] Derek Bruening and Qin Zhao. “Practical Memory Checking with Dr. Memory”. In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’11. Washington, DC, USA: IEEE Computer Society, 2011, pages 213–223. ISBN: 978-1-61284-356-8.
- [5] C99 Committee. *Rationale for International Standard–Programming Language–C. Revision 5.10*. 2003.
- [6] Vitaly Chipounov and George Candea. *Dynamically Translating x86 to LLVM using QEMU*. Technical report. École polytechnique fédérale de Lausanne, 2010.
- [7] Standard Performance Evaluation Corporation. *CINT2006 (Integer Component of SPEC CPU2006)*. Accessed October 2017. URL: <https://www.spec.org/cpu2006/CINT2006/>.
- [8] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. “FormatGuard: Automatic Protection From printf Format String Vulnerabilities.” In: *USENIX Security Symposium*. Volume 91. Washington, DC. 2001. URL: <https://www.usenix.org/legacy/events/seco1/cowanbarringer.html>.
- [9] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. “Buffer overflows: attacks and defenses for the vulnerability of the decade”. In: *DARPA Information Survivability Conference and Exposition, 2000. DISCEX ’00. Proceedings*. Volume 2. 2000, 119–129 vol.2. DOI: 10.1109/DISCEX.2000.821514.
- [10] Artem Dinaburg and Andrew Ruef. “Mcsema: Static translation of x86 instructions to llvm”. In: *ReCon 2014 Conference, Montreal, Canada*. 2014.

Introspection for C and its Applications to Library Robustness

- [11] Frank Ch Eigler. “Mudflap: Pointer Use Checking for C/C+”. In: *Proceedings of the First Annual GCC Developers’ Summit* (2003), pages 57–70.
- [12] Common Weakness Enumeration. *CWE-242: Use of Inherently Dangerous Function*. Accessed July 2017. 2017. URL: <https://cwe.mitre.org/data/definitions/242.html>.
- [13] Christof Fetzer and Zhen Xiao. “A flexible generator architecture for improving software dependability”. In: *13th International Symposium on Software Reliability Engineering, 2002. Proceedings*. 2002, pages 102–113. DOI: 10.1109/ISSRE.2002.1173221.
- [14] Christof Fetzer and Zhen Xiao. “An Automated Approach to Increasing the Robustness of C Libraries”. In: *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. DSN ’02. Washington, DC, USA: IEEE Computer Society, 2002, pages 155–166. ISBN: 0-7695-1597-5.
- [15] Cristof Fetzer and Zhen Xiao. “Detecting heap smashing attacks through fault containment wrappers”. In: *Proceedings 20th IEEE Symposium on Reliable Distributed Systems*. 2001, pages 80–89. DOI: 10.1109/RELDIS.2001.969756.
- [16] Narain Gehani. “Exceptional C or C with exceptions”. In: *Software: Practice and Experience* 22.10 (1992), pages 827–848. ISSN: 1097-024X. DOI: 10.1002/spe.4380221003.
- [17] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. “Memory-safe Execution of C on a Java VM”. In: *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*. PLAS’15. Prague, Czech Republic: ACM, 2015, pages 16–27. ISBN: 978-1-4503-3661-1. DOI: 10.1145/2786558.2786565.
- [18] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. “TypeSan: Practical Type Confusion Detection”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pages 517–528. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978405.
- [19] Reed Hastings and Bob Joyce. “Purify: Fast detection of memory leaks and access errors”. In: *In proc. of the winter 1992 usenix conference*. Citeseer. 1991.
- [20] Urs Hölzle, Craig Chambers, and David Ungar. “Debugging Optimized Code with Dynamic Deoptimization”. In: volume 27. 7. New York, NY, USA: ACM, July 1992, pages 32–43. DOI: 10.1145/143103.143114.
- [21] IEEE. “Systems and software engineering – Vocabulary”. In: *ISO/IEC/IEEE 24765:2010(E)* (Dec. 2010), pages 1–418. DOI: 10.1109/IEEESTD.2010.5733835.
- [22] Stephen Kell. “Dynamically Diagnosing Type Errors in Unsafe Code”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: ACM, 2016, pages 800–819. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2983998.

- [23] Stephen Kell. “Towards a Dynamic Object Model Within Unix Processes”. In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Onward! 2015. Pittsburgh, PA, USA: ACM, 2015, pages 224–239. ISBN: 978-1-4503-3688-8. DOI: 10.1145/2814228.2814238.
- [24] David G. Korn and Kiem-Phong Vo. “SFIO: Safe/Fast String/File IO”. In: *Proceedings of the Summer 1991 USENIX Conference, Nashville, TE, USA, June 1991*. 1991, pages 235–256.
- [25] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, pages 75–. ISBN: 0-7695-2102-9.
- [26] Peter A. Lee. “Exception handling in C programs”. In: *Software: Practice and Experience* 13.5 (1983), pages 389–405. ISSN: 1097-024X. DOI: 10.1002/spe.4380130502.
- [27] Kai Lu. *Analysis of CVE-2016-0059 - Microsoft IE Information Disclosure Vulnerability Discovered by Fortinet*. Accessed July 2017. 2016. URL: <https://blog.fortinet.com/2016/02/19/analysis-of-cve-2016-0059-microsoft-ie-information-disclosure-vulnerability-discovered-by-fortinet>.
- [28] Todd C. Miller and Theo de Raadt. “Strncpy and Strcat: Consistent, Safe, String Copy and Concatenation”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’99. Monterey, California: USENIX Association, 1999, pages 41–41. URL: <https://www.usenix.org/legacy/event/usenix99/millert.html>.
- [29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. “CETS: Compiler Enforced Temporal Safety for C”. In: *SIGPLAN Not.* 45.8 (June 2010), pages 31–40. ISSN: 0362-1340. DOI: 10.1145/1837855.1806657.
- [30] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C”. In: *SIGPLAN Not.* 44.6 (June 2009), pages 245–258. ISSN: 0362-1340. DOI: 10.1145/1543135.1542504.
- [31] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: ACM, 2007, pages 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746.
- [32] Manuel Rigger. “Sulong: Memory Safe and Efficient Execution of LLVM-Based Languages”. In: *ECOOP 2016 Doctoral Symposium*. Rome, Italy, 2016.

Introspection for C and its Applications to Library Robustness

- [33] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. “Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle”. In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2016. Amsterdam, Netherlands: ACM, 2016, pages 6–15. ISBN: 978-I-4503-4645-0. DOI: 10.1145/2998415.2998416.
- [34] Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. “Lenient Execution of C on a Java Virtual Machine or: How I Learned to Stop Worrying and Run the Code”. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. Prague, Czech Republic: ACM, 2017, pages 35–47. ISBN: 978-I-4503-5340-3. DOI: 10.1145/3132190.3132204.
- [35] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebe Jr. “Enhancing Server Availability and Security Through Failure-oblivious Computing”. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI’04. San Francisco, CA: USENIX Association, 2004, pages 21–21. URL: <https://www.usenix.org/legacy/events/osdio4/tech/rinard.html>.
- [36] John Rose. *JEP 243: Java-Level JVM Compiler Interface*. Accessed July 2017. 2014. URL: <http://openjdk.java.net/jeps/243>.
- [37] SANS. *CWE/SANS TOP 25 Most Dangerous Software Errors*. Accessed July 2017. 2011. URL: <https://www.sans.org/top25-software-errors/>.
- [38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. “AddressSanitizer: A Fast Address Sanity Checker”. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC’12. Boston, MA: USENIX Association, 2012, pages 28–28. URL: <https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany>.
- [39] Julian Seward and Nicholas Nethercote. “Using Valgrind to Detect Undefined Value Errors with Bit-precision”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’05. Anaheim, CA: USENIX Association, 2005, pages 2–2. URL: <https://www.usenix.org/legacy/events/usenix05/tech/general/seward.html>.
- [40] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. “Detecting Format String Vulnerabilities with Type Qualifiers”. In: *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*. SSYM’01. Washington, D.C.: USENIX Association, 2001. URL: <https://www.usenix.org/legacy/events/seco1/shankar.html>.
- [41] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wu Yang. “LLBT: An LLVM-based Static Binary Translator”. In: *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES ’12. Tampere, Finland: ACM, 2012, pages 51–60. ISBN: 978-I-4503-1424-4. DOI: 10.1145/2380403.2380419.
- [42] shootouts. *The Computer Language Benchmarks Game*. Accessed July 2017. URL: <http://benchmarksgame.alioth.debian.org/>.

- [43] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. “An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance”. In: *Proceedings of the 4th Workshop on Scala*. SCALA '13. Montpellier, France: ACM, 2013, 9:1–9:8. ISBN: 978-1-4503-2064-1. DOI: 10.1145/2489837.2489846.
- [44] Evgeniy Stepanov and Konstantin Serebryany. “MemorySanitizer: fast detector of uninitialized memory use in C++”. In: *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. San Francisco, CA, USA, 2015, pages 46–55. DOI: 10.1109/CGO.2015.7054186.
- [45] Bjarne Stroustrup. *The Design and Evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994. ISBN: 0-201-54330-3.
- [46] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. “SoK: Eternal War in Memory”. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pages 48–62. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.13.
- [47] TIOBE. *TIOBE Index for July 2017*. Accessed July 2017. 2017. URL: <http://www.tiobe.com/tiobe-index/>.
- [48] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. “Memory Errors: The Past, the Present, and the Future”. In: *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*. RAID'12. Amsterdam, The Netherlands, 2012, pages 86–106. ISBN: 978-3-642-33337-8. DOI: 10.1007/978-3-642-33338-5_5.
- [49] John Viega, J. T. Bloch, Yoshi Kohno, and Gary McGraw. “ITS4: a static vulnerability scanner for C and C++ code”. In: *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*. Dec. 2000, pages 257–267. DOI: 10.1109/ACSAC.2000.898880.
- [50] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. “Undefined Behavior: What Happened to My Code?” In: *Proceedings of the Asia-Pacific Workshop on Systems*. APSYS '12. Seoul, Republic of Korea: ACM, 2012, 9:1–9:7. ISBN: 978-1-4503-1669-9. DOI: 10.1145/2349896.2349905.
- [51] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. “Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farmington, Pennsylvania: ACM, 2013, pages 260–275. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522728.
- [52] *Whetstone benchmark*. Accessed July 2017. URL: <http://www.netlib.org/benchmark/whetstone.c>.
- [53] Christian Wimmer and Thomas Würthinger. “Truffle: A Self-optimizing Runtime System”. In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA, 2012, pages 13–14. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384723.

Introspection for C and its Applications to Library Robustness

- [54] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pages 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581.
- [55] Yves Younan, Wouter Joosen, and Frank Piessens. “Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs”. In: *ACM Comput. Surv.* 44.3 (June 2012), 17:1–17:28. ISSN: 0360-0300. DOI: 10.1145/2187671.2187679.
- [56] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. “APISan: Sanitizing API Usages through Semantic Cross-Checking”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pages 363–378. ISBN: 978-1-931971-32-4. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun>.

A Preliminary Performance Evaluation

The focus of this work was on evaluating the usefulness of exposing introspection functions to library writers. We have not yet invested much time in optimizing the peak performance of our approach in Safe Sulong. To demonstrate that Safe Sulong can run programs in a testing environment, we ran six benchmarks of the Computer Language Benchmark Game [42] (binarytrees, fannkuchredux, fasta, mandelbrot, nbody, and spectralnorm) and the whetstone benchmark [52], once with the enhanced libc and once without introspection checks. We determined the average peak performance of 10 runs by measuring the execution time after 50 in-process warm-up iterations. On these benchmarks, Safe Sulong’s peak performance was $2.3\times$ slower than executables compiled by Clang with all optimizations turned on (-O3 flag). We were unable to find any observable performance differences between the two libc versions, which is in part due to some of the introspection checks redundantly duplicating automatic checks performed by the JVM (e.g., bounds checks); such redundant checks can be eliminated by using the Graal compiler (e.g., through conditional elimination [43]). As part of future work, we will evaluate Safe Sulong’s performance in combination with the enhanced libc on larger benchmarks that stress the introspection functionality.

B Introspection Functions

Table 1 shows the functions and macros of the introspection interface. Internal functions that are private to the implementation are denoted with an underscore prefix.

Introspection for C and its Applications to Library Robustness

■ **Table 1** Functions and macros of the introspection interface

Object bounds functions		
<code>long _size_right(void *)</code>	Primitive internal	Returns the space in bytes from the pointer target to the end of the pointed object. This function is undefined for illegal pointers.
<code>long _size_left(void *)</code>	Primitive internal	Returns the space in bytes from the pointer target to the beginning of the pointed object. This function is undefined for illegal pointers.
<code>long size_right(void *)</code>	Composite	Returns the remaining space in bytes to the right of the pointer. Returns -1 if the pointer is not legal or out of bounds.
<code>long size_left(void *)</code>	Composite	Returns the remaining space in bytes to the left of the pointer. Returns -1 if the pointer is not legal or out of bounds.
Memory location functions		
<code>Location location(void *)</code>	Primitive	Returns the kind of the memory location of the referenced object. Returns -1 if the pointer is NULL.
<code>bool freeable(void *)</code>	Composite	Returns whether the pointer is freeable (i.e., DYNAMIC non-null memory; pointer referencing the beginning of an object).
Type functions		
<code>void* try_cast(void *, struct Type *)</code>	Primitive	Returns the first argument if the pointer is legal, within bounds, and the referenced object can be treated as of being of the specified type and NULL otherwise.
Variadic function macros		
<code>int count_varargs()</code>	Primitive	Returns the number of variadic arguments that are passed to the currently executing function.
<code>void* _get_vararg(int i)</code>	Primitive internal	Returns the i^{th} variadic argument (starting from 0) and returns NULL if i is greater or equal to <code>count_varargs()</code> .
<code>void* get_vararg(int i, Type* type)</code>	Composite	Returns the i^{th} variadic argument (starting from 0) as the specified type. Returns NULL if the object cannot be treated as being of the specified type or if i is greater or equal to <code>count_varargs()</code> .

About the authors

Manuel Rigger is a PhD student at Johannes Kepler University Linz, Austria. His main research interests are low-level software security, run-time detection of bugs in C, and language implementation. Currently, he works on the safe execution of low-level languages on the Java Virtual Machine. Contact him at manuel.rigger@jku.at.



René Mayrhofer heads the Institute of Networks and Security at Johannes Kepler University Linz, Austria, and the Josef Ressel Center on User-friendly Secure Mobile Environments (u'smile). His research interests include computer security, mobile devices, network communication, and machine learning, which he brings together in his research on securing spontaneous, mobile interaction. Contact him at rene.mayrhofer@jku.at



Roland Schatz is a senior researcher at Oracle Labs. His research interests are programming languages, virtual machines and dynamic compilation. His current research focus is cross-language interoperability between dynamic and static languages. He received his PhD in Computer Science from Johannes Kepler University in 2013. Contact him at roland.schatz@oracle.com.



Matthias Grimmer is a senior researcher at Oracle Labs. His research focuses on compilers and virtual machines. Grimmer received his PhD in Computer Science from Johannes Kepler University in 2015. His PhD thesis is titled “Cross-language interoperability in a multi-language runtime”. Contact him at contact@matthiasgrimmer.com.



Hanspeter Mössenböck is a professor of Computer Science at the Johannes Kepler University in Linz, Austria, and the head of the Institute for System Software. He mainly works on programming languages, compilers, and virtual machines with a special focus on dynamic compiler optimizations. Other areas of interest include application performance monitoring, software visualization, as well as static and dynamic program analysis. Contact him at hanspeter.moessenboeck@jku.at.

